04

Método abstracto

Transcripción

[00:00] Bien. Una de las cosas que tenemos que ver aquí muy bien es que todos los beneficios del polimorfismo continúan disponibles como ya lo hemos visto. Podemos seguir instanciando las clases, los objetos que extienden de funcionario, los beneficios a nivel ya de métodos y ejecución, todo eso sigue igual de disponible. ¿Entonces cuál es la verdadero ganancia de volver una clase abstracta?

[00:33] Vamos a ponerlo de otra forma. Ya que funcionario digamos es una entidad abstracta, ¿no debería tener una propia regla para la bonificación, que sería getBonificación? Habíamos quedado que cada objeto, el gerente define su propia bonificación y el contador define su propia bonificación, y métodos sobreescritos.

[01:04] Entonces el funcionario no debería tener pues ninguna bonificación porque es algo abstracto, cada uno va a definir su propia bonificación. Entonces prácticamente no necesitamos ya más este método de aquí, lo que hacemos es borrarlo, vamos a darle aquí y vamos a ver qué pasaría si nosotros borramos ese método. Y automáticamente vemos que explotó un error aquí, aquí y aquí.

[01:30] Vamos a entrar al gerente. En el caso del gerente él me está dando un error. ¿Por qué? Porque él llamaba a getBonificacion del funcionario. Esto tranquilamente puede ser reemplazado por lo que era la bonificación del funcionario que era this.getSalario por 0.5, habíamos quedado. No, por 0.05. Y problema resuelto, no tenemos nada más que hacer, el gerente sigue

definiendo su propia bonificación y no interfiere con lo demás. Así que guardamos aquí. Vemos que se fue el error.

[02:08] Aquí, en contador. En contador, el problema que estamos teniendo es lo que vimos en el video anterior, perdón, en la clase anterior sobre override, ya que override no está sobreescribiendo ningún método getBonificacion porque no existe. Entonces, ¿qué hacemos? Lo borramos y el código compila correctamente.

[02:31] Nuevamente, override lo que hacía era explicar que estamos sobreescribiendo este método, pero si este método no existe en funcionario, entonces no tiene por qué existir, solamente borramos la anotación y asunto arreglado. Aquí tenemos un problema un poco más grande. ¿Cuál es?

[02:50] Que registrar salario recibe como parámetro a funcionario. Hasta ahí todo bien. ¿Pero dónde está el detalle? ¿Dónde está nuestra prueba mayor? Que para él sumar las bonificaciones, él usa el método getBonificacion de funcionario. ¿Cómo podemos resolver este problema? Bueno, creando un registrar salario para gerente, otro para contador y así conforme vayamos creando nuevos cargos en la empresa, lo cual ya vimos en dos cursos anteriores, creo, que no tiene el menor sentido.

[03:29] Entonces la conclusión es no podemos eliminar el método getBonificación de funcionario porque aquí nos va a causar un problema tremendo y nos va a perjudicar mucho en el diseño de nuestro sistema. No es bueno borrar el método getBonificacion. Vamos a funcionario y vamos a darle un "Ctrl + Z". Listo, tenemos nuevamente nuestro método getBonificación.

[03:56] Pero nuevamente llego a mi problema anterior. Pero yo no deseo que este método tenga una implementación, yo no necesito nada de esto. Ahora, si no lo necesito, tranquilamente puedo ignorarlo. No hay ningún problema en ignorar la implementación y que cada uno así escriba su propio método. Pero

estamos muy propensos y muy expuestos a errores humanos por decirlo de alguna forma.

[04:27] Porque estamos dejando carta abierta a que tomen este código por defecto. Y si por ejemplo decimos: "Ah, entonces retornamos cero", el problema sigue existiendo. ¿Por qué? Porque si alguien no hace esta implementación, entonces él va a retornar cero de bonificación y puede que cause otro tipo de problemas.

[04:50] Recuerden cuál es el objetivo de un sistema. Es facilitar la vida a todos los trabajadores o todas las personas que interactúan con él y reducir el trabajo. Es más que nada eso, reducir el trabajo, entonces nosotros, al dejar esta responsabilidad de lado del desarrollador, de otras personas, estamos aumentando el trabajo de ellos, la responsabilidad que ellos tienen sobre el código y eso no está bien.

[05:21] Entonces si no queremos que este método tenga una implementación, si solamente lo comentamos nos va a dar un error porque necesitamos retornar alguna cosa, pero aquí entra otro beneficio de la clase abstracta y es que nosotros también podemos definir métodos abstractos según la clase abstracta. ¿Cómo es eso?

[05:44] Si no quieres que tu método tenga una implementación, que sería esto, lo que podemos hacer es borramos el cuerpo totalmente, dejamos el método ahí y lo dejamos listo para que simplemente sea implementado por las clases hijas. ¿Cómo hacemos eso? Volvemos al método también abstracto.

[06:15] Y ya está. Vemos que el método abstracto automáticamente ya está listo, compila, no necesita definir un cuerpo porque si yo le defino un cuerpo entonces me va a dar aquí un error de compilación. ¿Por qué? Porque al ser abstracto, siguiendo el mismo concepto de la clase abstracta que ya hemos dicho que conceptualmente existe pero no existe realmente, en el método es igualito.

[06:45] Existe conceptualmente, yo tengo un método getBonificacion conceptualmente pero realmente él no tiene implementación, no existe, es un método libre. Le doy punto y coma aquí y ya tengo mi método abstracto getBonificacion. Y este método obliga a todas las clases hijas a sobreescribir ese método. Por ejemplo, si yo aquí le doy a un getBonificacion, guardo aquí, automáticamente él deja de compilar aquí. ¿Por qué?

[07:17] Porque aquí me dice que el tipo Gerente debe implementar el método abstracto getBonificacion. Y aquí me da una ayuda de implementar el método. Yo ya lo tengo aquí en este caso, está aquí. Entonces lo que voy a hacer es volver a la normalidad, vemos que compila, guarda correctamente y el código sigue funcionando aquí.

[07:45] Y en controlBonificacion pues ya no tenemos ningún tipo de problema porque funcionario ya tiene el método getBonificacion, solo que no está implementado. Eso es un beneficio a tener la clase abstracta. Vemos que desaparecieron los problemas de las demás clases también. ¿Por qué? Porque contador también está implementando el método getBonificacion.

[08:15] De igual forma con gerente, si yo borrara este método de aquí, entonces el compilador no me va a dejar seguir. ¿Por qué? Porque me va a decir: "no has implementado el método getBonificación de funcionario". Recuerden que todo método abstracto, de reglas abstractas, tiene que ser implementado por la clase que está extendiendo esa clase abstracta.

[08:35] Si le doy agregar método no implementado me va a generar el código igual y voy a poder retornar pues mi lógica de negocio. Aquí era 200, borro esto y vemos que nuevamente solamente agregó la anotación override para asegurarme que yo estoy sobreescribiendo el método correcto y de ahí todo sigue su curso sin ningún problema.

[08:59] ¿En qué sucedería si yo por ejemplo no quiero que el contador implemente el método getBonificación? ¿Qué puede ser? Bueno, en ese caso, si

la clase es normal digamos, es una clase común y corriente, yo estoy obligado a implementar dos métodos abstractos. Pero si la clase es abstracta, aquí, entonces, yo no soy obligado a implementar los métodos abstractos de getFuncionario.

[09:32] Pero al mismo tiempo yo ya no puedo instanciar al contador. ¿Por qué? Porque es abstracto. Entonces ese es uno de los beneficios que nos da la clase abstracta. Podemos crear métodos abstractos y hacer que nuestra orientación a objetos quede mucho más entendible y parecida con el mundo real que queremos representar en este sistema.

[10:00] Funcionario ya existe como concepto abstracto, todos somos funcionarios en nuestras empresas, y bonificación va a ser calculada por cada tipo de funcionario porque el método es abstracto.